# PUZZLER: An Automated Logic Puzzle Solver

Aleksandar Milicevic, Joseph P. Near, and Rishabh Singh

Massachusetts Institute of Technology (MIT)

**Abstract.** Solving logic puzzles is a difficult problem, requiring the parsing of general phrases, semantic interpretation of those phrases, and logical inference based on the resulting information. Moreover, since human reasoning is generally unsound, extra background information must be generated to allow for the logical inference step—but without introducing facts that would result in an incorrect answer.
We present PUZZLER, a tool that automatically solves logic puzzles described using informal natural language. Our method translates the puzzle's description to a formal logic, infers appropriate background information, and automatically finds a solution. We show that PUZZLER can solve the classic Einstein puzzle, and our solution is generally applicable to other logic puzzles.

## 1 Introduction

Recent research in natural language processing has been directed towards techniques using statistical methods [2]. Statistical methods alone, however, may not allow computers to replicate many of the language tasks we perform daily. Solving a logic puzzle requires a deep understanding of the underlying meaning of the puzzle. The puzzle's solution is not present in its superficial description, but rather must be inferred. A similar deep understanding is required in many other NLP applications, such as Question Answering (QA), Information Extraction (IE), Summarization, Machine Translation and Paraphrasing, and Information Retrieval (IR). Logic puzzles thus represent an interesting research domain, as they allow for the exploration of the issues involved without the complexities of larger applications.

Logic puzzles are easy for humans to understand, but very difficult for computers—precisely because they require both semantic interpretation and the inference of background knowledge. This background knowledge reflects the fact that human reasoning is often unsound: it is often impossible to logically derive the solution to a puzzle using *only* the information provided. Moreover, the necessary background knowledge is very difficult to infer, since it is based on the knowledge base humans build up over years of learning. Logic puzzles have therefore been studied [5, 6], but whether a general method exists for finding their solutions still remains an open question.

We present PUZZLER, a step towards that solution. PUZZLER is a general tool for the translation of logic puzzles into formal logic, and is also capable of

producing solutions consistent with those translations. Our results are preliminary, but as evaluation on the classic Einstein puzzle shows, PUZZLER is already capable of translating and solving real puzzles. PUZZLER is not a complete solution, however, since it cannot infer all of the necessary background information: the user is still required to specify the domain of the problem.

## 2 Logic Puzzles

A logic puzzle generally consists of a set of natural-language rules representing constraints on the set of solutions to the puzzle, and a query to which the correct answer represents a proof that the user has solved the puzzle correctly. Puzzles are typically designed so that only a single solution satisfies all of the puzzle's constraints, and thus there is only one possible answer to the puzzle's query.

The solver is expected to use the information present both in the constraints of the puzzle and in its query in preparing a solution. In addition, a large amount of background knowledge may be expected of the solver. He or she may be expected to know, for example, that Old Gold is a brand of cigarette, or that coffee is a drink. Moreover, assumptions about the multiplicity of relationships between entities in the puzzle are common. Most puzzles assume that the mappings between entities are one-to-one, but this is not universally the case; once again, the solver must make use of background knowledge to decide.

Soving logic puzzles is difficult for humans because we cannot efficiently solve constraint satisfaction problems over domains of any significant size. Interpreting the problem, on the other hand, is rarely difficult: we tend to make the same assumptions and have access to the same background knowledge as the author of the puzzle. For the computer, this situation is reversed: a machine can solve constraint satisfaction problems fairly efficiently, but does not have the same background knowledge that a human does.

### 2.1 Einstein's puzzle

The Einstein puzzle (also known as the Zebra puzzle) is perhaps the most famous logic puzzle. Albert Einstein is rumored to have invented it in the late 1800s, but there is no real evidence for this theory; the version reproduced below was published in the Life International magazine in 1962 [1].

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

Formally speaking, the rules of the puzzle leave out many details. For example, there is a total ordering on houses, but this fact is not stated in the puzzle. Instead, we infer it based on the fact that houses are usually built next to one another. Neither the zebra nor water is mentioned in the clues; the clever solver could therefore answer "nobody" to both questions, but the generally accepted answer calls for the inference that *someone* owns the zebra, and *someone* drinks water.

### 2.2 Background Information

Formal logic allows the proof of a statement only if it follows directly from the information given. This property is called *soundness*, and is considered desirable in formal logics. Much of the reasoning done by humans, on the other hand, is *unsound*—we tend to accept the most likely explanation for a situation, even if we cannot show using the information we have that the explanation is correct. The addition of background information to a formal model of a real-life situation can allow for the sound inference of the correct answer in situations that normally would require unsound inferences.

The background information associated with the Einstein puzzle is indicative of the importance of this background information in general. To solve the puzzle using sound logical inference, we must also have the following facts, even though the puzzle gives no reason that they should hold:

1. Every person lives in one house.
2. Every person drinks one drink.
3. Every person owns one pet.
4. Every person has one citizenship.
5. Every person smokes one brand.
6. Every house has one color.
7. There exists a total order on houses.

Implicit in these facts is a segregation of the entities in the puzzle into categories, or *types*. For example:

1. Every Englishman is a person.
2. Every dog is an animal.
3. Red is a color.

4. Lucky Strike is a brand.
5. Tea is a beverage.

Automatic inference of this background and type information is known to be a difficult task. While it seems natural to relax slightly the soundness constraints on logical reasoning, doing so arbitrarily results in inferences that would be considered silly, even by humans. Distinguishing between reasonable and unreasonable assumptions is thus very tricky.

### 2.3 Semantic Translation and Solving

Logic puzzles are typically written in a style requiring the expressive power of first-order logic. This presents no problem to the human solver, as he or she generally has little trouble reducing the problem to that of constraint satisfaction, given the small finite domain. Automated solvers for first-order logic, however, tend to focus on first-order theorems over unbounded domains, and due to the undecidability of first-order logic, may fail to terminate.

While an appropriate automated solver is an issue orthogonal to that of semantic analysis, it is nevertheless an important part of the overall solution to the problem.

## 3 Approach

Puzzler combines several existing NLP tools with our own semantic translation and background information inference engine. Each part of Puzzler is designed to be as general as possible, and does not rely on any information specific to the Einstein puzzle. Figure 1 summarizes the architecture of Puzzler.

Puzzler is composed of the Link Grammar [7] general-purpose English parser, a semantic translator, and an automated logical analyzer. Our semantic translator makes use of the structure of the parse trees generated by the parser, along with semantic information produced by the WordNet [3] database; it is thus also general-purpose. For logical analysis, we employ the Alloy Analyzer [4], which solves general first-order logic problems.

### 3.1 Parsing

In order to make our solution as general as possible, we make use of a general-purpose, state-of-the-art English parser. This choice allows us the flexibility to generalize to new types of logical statements, as our parser is not specialized to any single problem. Previous work (e.g. [5]) made use of specialized parsers to obtain important information about problem instances, restricting their solutions to particular sets of problems.

We employ Sleator et. al's Link Grammar parser [7] to perform sentence parsing. In addition to being robust, reasonably precise, and accurate, the Link

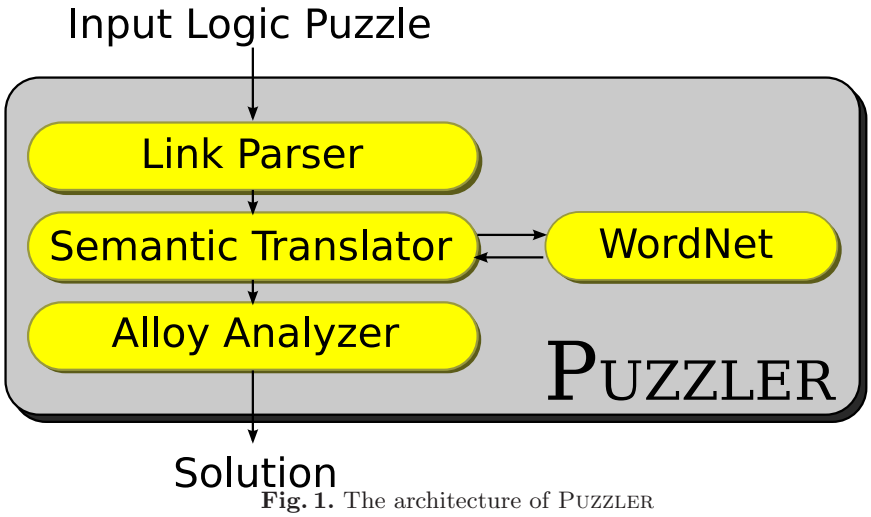


**Fig. 1.** The architecture of PUZZLER

parser is small, very fast, and can produce parse trees for a very wide variety of English entences.

In our experience, the Link parser easily parses most straightforward logical sentences of the form usually found in logic puzzles. Occasionally, however, the parser is confused by ambiguous uses of words that can be assigned more than one part of speech. The sentence "Coffee is drunk in the green house," for example, is taken from the original Einstein puzzle, and is incorrectly parsed by the Link parser. The word "drunk" is taken to be an adjective, resulting in a parse tree whose semantics involve coffee beans having had too much to drink. In these cases, the user must rephrase the English sentence to avoid the ambiguity. While this kind of mistake is inevitable when a generalized parser is used, we found the flexibility gained through the parser's generality to outweigh its drawbacks; moreover, when parse errors are made, our semantic translation can often discover that the resulting parse tree is flawed, and notify the user of the unresolvable ambiguity in the problem.

### 3.2 Semantic Translation: Facts

Our fact translation transforms parse trees produced by the Link parser into unambiguous logical formulas. Our translation relies on the correctness of the parse trees produced by the Link parser, but it is also able to recover from some of the typical mistakes that the parser may commit. Sometimes it happens that a prepositional phrase that should be linked lower than the corresponding noun phrase is actually linked at the same level as the noun phrase or maybe even one level above. Here is a concrete example from the Zebra puzzle: the sentence

*Kools are smoked in the house next to the house where the horse is kept* is parsed by the Link parser as

```
(S (NP Kools)
   (VP are
       (VP smoked
           (PP in
               (NP the house))
           (PP next to
               (NP the house))
           (PP where
               (S (NP the horse)
                  (VP is
                      (VP kept))))))
   .)
```

instead of as

```
(S (NP Kools)
   (VP are
       (VP smoked
           (PP in
               (NP the house))
                   (PP next to
                       (NP the house))
                       (PP where
                           (S (NP the horse)
                               (VP is
                                   (VP kept))))))
   .)
```

However, because of the generic design of our translation, this de-facto incorrect parse tree produced by the Link parser will not cause too much trouble. The translation of the first two of the three parallel prepositional phrases will produce a logical expression each, both of which evaluate to some house, and the translation of the third one will produce an expression that evaluates to a horse. Afterwards, it is left to somehow semantically interconnect (relate) those expressions. In the case of the correct parse tree, the translator doesn't have a choice, the horse has to be related to the house, and then that house is related to another house, and finally that other house is related to Kools. In the case when the incorrect parse tree is produced by the parser (the one with the three parallel prepositional phrases), our procedure relates all three expressions in a sequential order which is in this case is semantically correct.

A more detailed description of the concrete translation rules we employ is given in the Solutions section.

### 3.3 Semantic Translation: Background Information

In order to avoid errors, we take a conservative approach to the addition of background information. When the puzzle contains words such as "next" and "previous," for example, we infer that there exists an ordering on the types of the nouns these words modify. There are several other such special words that we know how to interpret. Those words include "first", "last", "second", "middle", "left", "right", etc.

There is some background information that our system cannot infer. While we know how to infer when an ordering on a type is necessary, inferring the types themselves is much more difficult. Our current implementation requires the user to enumerate the types present in his or her problem, and to specify the nouns representing members of each type. We use WordNet [3] in a limited capacity to transform nouns into their singular forms, freeing the user from specifying both singular and plural forms of each type member; we imagine that WordNet's classification facilities might be useful in inferring some information both about the types present in a given problem and about their members. This solution would make our technique completely automated, and is discussed in detail later.

### 3.4 Alloy

Alloy [4] is a formal language for modelling software designs. The language comprises first-order relational logic with transitive closure, written in a flexible and concise notation. Modelling in the Alloy language is supported by the Alloy Analyzer, a fully automatic bounded analysis tool and visualizer. Using the Alloy Analyzer, a modeller can examine the behavior of his or her model, check properties of the model, and produce informative graphs of the model's instances. The Alloy language has been used to model data structures, concurrent systems, programming patterns, and even logic puzzles; the Alloy Analyzer allows for the automated analysis of these models, including the solving of properly encoded puzzles.

The Analyzer reduces an undecidable analysis to an NP-complete one by considering only a bounded number of objects of each type. This approach allows an exhaustive search to be conducted for a counterexample to the user's claim. This analysis is performed by the Kodkod [8] relational model finder via a reduction to the boolean satisfiability problem. While SAT is an NP-complete problem, modern solvers tend to perform well on the instances produced by Kodkod, allowing for a fully automated analysis of Alloy models.

The Alloy Analyzer gives us a very appropriate tool with which to solve logic puzzles. In contrast to general theorems in first-order logic, logic puzzles—like many other interesting first-order logic problems—tend to be defined over finite domains. The Einstein puzzle, for example, describes a fixed number of each entity. For these problems, the Alloy Analyzer's bounded analysis is a perfect match; since first-order logic is the generally-accepted standard for semantic interpretation of natural language, Alloy is a good fit as a target language for the standard translation techniques.

## 4 Solution

We now present two translations of logic puzzles into Alloy, followed by the set of rules used by Puzzler to arrive at a correct translation. Our first translation is natural, but is not easily automated; the second contains redundant relations that allow for a simpler translation. Finally, we present the set of translation rules used to arrive at the simpler translation.

For simplicity, we present these translations using a smaller puzzle generated from `http://www.mensus.net/brain/logic.shtml?code=3FFE.287645C`:

1. The Danish lives in the third house.
2. The third house is black.
3. The Irish lives directly next to the pink house.
4. The Danish lives directly to the right of the espresso drinking person.
5. The person drinking beer lives directly to the right of the red house.

It is also explicitly given that there are three different nationalities (`Danish`, `Swiss`, `Irish`), three different house colors (`Red`, `Black`, `Pink`), and three different drinks (`Espresso`, `Wine`, `Beer`)

The question is: *Where does everybody live?*

### 4.1 Translation to Alloy

The most natural Alloy model of this puzzle defines a `Sig` for each type in the puzzle (`Nationality`, `Color`, `Drink` and `House`) and concrete instances of each `Sig`:

```
abstract sig Nationality {}
one sig Danish, Swiss, Irish extends Nationality {}

abstract sig House {}
one sig H1, H2, H3 extends House {}

abstract sig Drink {}
one sig Espresso, Wine, Beer extends Drink {}

abstract sig Color {}
one sig Red, Black, Pink extends Color {}
```

Next, the model should define some relations between domains. In most logic puzzles, it is sufficient to map elements of one domain (say `Nationality`) to elements of the others. Here, we define relations `lives`, `drinks`, and `house_color` to represent these mappings.

```
abstract sig Nationality {
  lives: one House,
  drinks: one Drink,
  house_color: one Color,
}
```

The keyword `one` is used to express the implicit constraint that every man lives in exactly one house, drinks one drink and has only one house color (these constraints come from our background knowledge, as discussed earlier). This keyword, however, does not make the appropriate relation one-to-one; without

any additional constraints, one potential solution to the puzzle involves every man drinking beer. To achieve the intended solution, we introduce still more background knowledge:

```
fact {
  all h: House | one h.~lives
  all d: Drink | one d.~drinks
  all c: Color | one c.~house_color
}
```

`~` is an operator that transposes the given relation. For example, if `lives` is a relation from `Nationality` to `House`, then the `~lives` is a relation from `House` to `Nationality` and relates the same objects as the `lives` relation.

There is another important piece of the background knowledge that can be easily expressed in Alloy: *ordering*. From the puzzle description given above, it is clear that the set of houses is totally ordered. This information allows some puzzle statements to talk about houses that are next to each other, a house that is immediately to the right of some other house, *etc.* To model this in Alloy, we use the *ordering* utility package by writing `open util/ordering[House] as houseOrd`. The variable named `houseOrd` can then be used to conveniently express most of the ordering-related constraints.

The last thing we need to do is to translate the puzzle statements into Alloy *facts*. We start with the first one: there are exactly two houses before the house in which the Danish lives:

```
fact { #houseOrd/prevs[Danish.lives] = 2} }
```

`Danish.lives` (a relational join) yields the house in which the Danish lives; `houseOrd/prevs` yields all previous houses. The hash operator (`#`) allows us to constrain the number of such houses.

The next statement says that the third house is black. Since there is no direct relation between houses and color, we relate the man living in the third house with the color `Black` (using the `house_color` relation). We achieve this by performing an additional relational join operation, which we call an *intermediate hop*:

```
fact { one h: House | #houseOrd/prevs[h]=2 ∧ h.~lives.house_color=Black}
```

We present the other statements more briefly. To say that a house is immediately to the left or right of another house, we use `houseOrd/prev[h1]=h2` or `houseOrd/next[h1]=h2`. Similarly, to encode the fact that a house is next to another house we say that it is either immediately to the left of immediately to the right of the other house. The remaining three statements are translated to these Alloy facts:

```
fact {
  Irish.lives in {houseOrd/prev[Pink.~house_color.lives] ∪
                  houseOrd/next[Pink.~house_color.lives]}
  houseOrd/prev[Danish.lives] = Espresso.~drinks.lives
  houseOrd/prev[Beer.~drinks.lives] = Red.~house_color.lives
}
```

The solution to this model produced by Alloy is unique, and is shown in Figure 2.

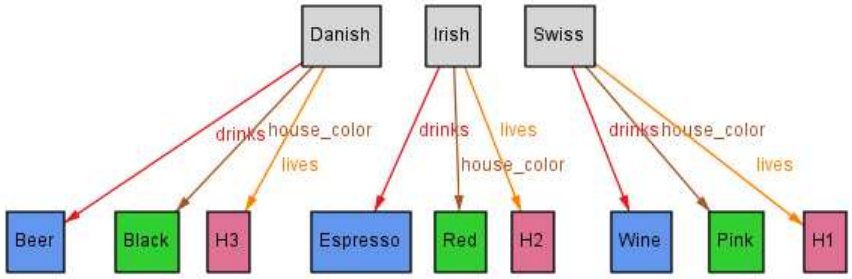


**Fig. 2.** Alloy solution for the example puzzle

### 4.2 Alternative Translation to Alloy

As illustrated in the previous section, to translate puzzle statements that talk about two unrelated domains (between which there is no direct relation), an intermediate hop must be introduced. Automating such a translation is difficult, because the translator itself must reason about how to build expressions. To simplify the translation, we add redundant relations between every pair of types.

Since these additional relations should not hold any new information, we must also introduce additional constraints to maintain synchronization between the redundant relations. Since these constraints depend only on the explicitly-given domains of the puzzle, however, they are simple to generate mechanically.

Consider a relation between `Nationality` and `House` (`nationality_house`), one between `House` and `Color` (`house_color`), and one between `Nationality` and `Color` (`nationality_color`). Then, if nationality $n_1$ and house $h_1$ are in the relation `nationality_house`, and house $h_1$ and color $c_1$ are in the relation `house_color`, we must ensure that nationality $n_1$ and color $c_1$ are also in the relation `nationality_color`.

This type of constraint is simple to express in Alloy. We present the alternative translation, which does not contain intermediate hops, below.

```
open util/ordering[House] as houseOrd

abstract sig Nationality {
  nationality_house : one House,
  nationality_drink : one Drink,
  nationality_color : one Color,
} {
}

one sig Danish, Swiss, Irish extends Nationality {}

abstract sig House {
  house_drink : one Drink,
  house_color : one Color,
  next_to : set House,
} {
  one this.~nationality_house
  one x: Nationality | x.nationality_house = this ∧
    x.nationality_drink=this.@house_drink ∧
```

```
    x.nationality_color=this.@house_color
  this.@next_to = { houseOrd/next[this] ∪ houseOrd/prev[this] }
}

one sig H1, H2, H3 extends House {}

abstract sig Drink {
  drink_color : one Color,
} {
  one this.~nationality_drink
  one this.~house_drink
  one x: Nationality | x.nationality_drink = this ∧
    x.nationality_color=this.@drink_color
}

one sig Espresso, Wine, Beer extends Drink {}

abstract sig Color {
} {
  one this.~nationality_color
  one this.~house_color
  one this.~drink_color
  one x: Nationality | x.nationality_color = this
}

one sig Red, Black, Pink extends Color {}

fact {
  #houseOrd/prevs[Danish.nationality_house] = 2}
  one h: House | #houseOrd/prevs[h]=2 ∧ h.house_color=Black
  Irish.nationality_house in {houseOrd/prev[Pink.~house_color] ∪
                              houseOrd/next[Pink.~house_color]}
  houseOrd/prev[Danish.nationality_house] = Espresso.~house_drink
  houseOrd/prev[Beer.~house_drink] = Red.~house_color
}
```

# 5 Implementation

## 5.1 Translation Procedure

Our translation procedure is based mostly on the interpretation of noun phrases. We decided to search only for noun phrases and try to properly interconnect (relate) them without examining the verb that connects them. Semantically, this is the correct thing to do because we know that in Einstein-like puzzles any two objects from the known set of domains can be related to each other. Our decision to have a relation between every two domains in the Alloy model helps simplify the implementation.

For example, the fact that the Danish lives in the third house can be said in any of the following ways:

1. The Danish lives in the third house.
2. The Danish is in the third house.
3. The Danish owns the third house.
4. The third house is owned by the Danish.
5. The third house hosts the Danish.

In all of these cases, we have the Danish on one side and the third house on the other, and all we need to do is to connect them by saying that the tuple `Danish, the third house` is in the relation `nationality_house`.

First of all, we invoke the Link parser to obtain parse trees of the input sentences. Then, we traverse the parse tree of the given input sentence in the depth-first fashion and for every traversed node in the tree we build and return a structure that contains the following data about that part of the sentence:

```
class Struct {
  String fact;
  Sig type;
  String varName;
  List<String> pps; // prepositional phrases discovered on the way
}
```

This way, we create the final fact that corresponds to the whole sentence in the bottom-up style, building up from the smaller pieces.

### 5.2 Pattern Matching

We begin by establishing the syntax for writing the pattens we will consider:

`_-` : one level lower in the parse tree hierarchy
`_` : at the same level in the parse tree hierarchy
`ANY` : matches any part of speech (POS) tag
`ANNY` : matches any number of parallel POS tags
`T*` : matches any number of parallel `T` tags

The most important pattern rules are explained in Figure 3

The core part of the algorithm that is left to be explained is the *relate* procedure. This procedure takes two structures, $s_1$ and $s_2$ and returns a new structure that relates both of them. If one of the inputs is null, it returns the other one. Otherwise, it tries to combine the two facts that correspond to these two structures using only prepositions collected for the structure $s_2$. Prepositions collected for the structure $s_1$ are propagated in the resulting structure that this procedure returns. If no prepositions are collected for $s_2$ or none of the collected prepositions can be interpreted by the embedded background knowledge database, it combines the two structures directly by returning the fact that looks like:

$s_1^{fact}$ && $s_2^{fact}$ && $s_1^{varName}.relation\_name = s_2^{varName}$.

As an example of the translation procedure, we give in Figure 4 a complete trace of our translation implementation for several sentences from the Einstein puzzle.

## 6 Evaluation

We evaluated Puzzler on the following three puzzles:

- The original Einstein puzzle
- A modified (less ambiguous) Einstein puzzle
- A randomly generated puzzle in the same style

| Pattern | Description | Action |
|---|---|---|
| NP__ | noun phrase with no children | Iterates in reverse over all words that this noun phrase consists of and looks for words that are either domain/instance names or words that are supported by our background knowledge database. As soon as it finds a domain, it declares that domain to be the return type of this noun phrase, and all other words are treated as its properties. |
| NP__(ADJP)* | noun phrase with any number of adjectives | all adjectives are treated as properties of the enclosing noun phrase |
| PP__ANY | prepositional phrase with exactly one sub-node | Traverses the child node and adds the prepositions from this prepositional phrase to the list of prepositions in the structure obtained by traversing the child node and returns that structure. |
| ADJP__ | adjective with no children | Gets the same treatment as a noun phrase with no children |
| VP__ | verb phrase with no children | Simply returns an empty structure. |
| VP__ANNY | verb phrase with any number of children | Traverses all children nodes and sequentially *relates* two by two of the resulting structures. |
| VP__NO_VP | any node having a noun phrase followed by a verb phrase as children | Traverses both noun phrase and verb phrase and *relates* the resulting structures. |
| ANY__WHNP_ANY | any node having a pronoun followed by anything as children | Ignores the pronoun and returns the result of the traversal of the other child node. |
| ANY__ANY | any node having exactly one sub-node | Traverses the child node and returns the result. |

**Fig. 3.** Description of the most important patterns

```
The green house is immediately to the right of the ivory house .
(S (NP The green house) (VP is (ADVP immediately) (PP to (NP (NP the right) (PP of (NP the ivory house))))) .)
##|  Pattern: _visit_S__NP_VP, Invoking: _visit_ANY__NP_VP
##|  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |result: one h9: House | h9.~color_house=Green, pps = []
##|  |  Pattern: _visit_VP__ADVP_PP, Invoking: _visit_VP__ANNY
##|  |  |  Pattern: _visit_ADVP__, Invoking: _visit_ADVP__
##|  |  |  |result: , pps = []
##|  |  |  Pattern: _visit_PP__NP, Invoking: _visit_PP__ANY
##|  |  |  |  Pattern: _visit_NP__NP_PP, Invoking: _visit_NP__NP_PP
##|  |  |  |  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |  |  |  |result: , pps = []
##|  |  |  |  |  Pattern: _visit_PP__NP, Invoking: _visit_PP__ANY
##|  |  |  |  |  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |  |  |  |  |result: one h10: House | h10.~color_house=Ivory, pps = []
##|  |  |  |  |  |result: one h10: House | h10.~color_house=Ivory, pps = [of]
##|  |  |  |  |result: one h10: House | h10.~color_house=Ivory, pps = [of, the, right]
##|  |  |  |result: one h10: House | h10.~color_house=Ivory, pps = [of, the, right, to]
##|  |  |result: one h10: House | h10.~color_house=Ivory, pps = [of, the, right, to]
##|  |result: one h9:House | h9.~color_house=Green && one h10:House | h10.~color_house=Ivory&&houseOrd/prev[h9]=h10, pps = []
one h9: House | h9.~color_house=Green && one h10: House | h10.~color_house=Ivory && houseOrd/prev[h9] = h10

Milk is drunk in the middle house .
(S (NP Milk) (VP is (ADJP drunk (PP in (NP the middle house)))) .)
##|  Pattern: _visit_S__NP_VP, Invoking: _visit_ANY__NP_VP
##|  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |result: one d15: Milk | 0==0, pps = []
##|  |  Pattern: _visit_VP__ADJP, Invoking: _visit_VP__ANNY
##|  |  |  Pattern: _visit_ADJP__PP, Invoking: _visit_ANY__ANY
##|  |  |  |  Pattern: _visit_PP__NP, Invoking: _visit_PP__ANY
##|  |  |  |  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |  |  |  |result: one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16], pps = []
##|  |  |  |  |result: one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16], pps = [in]
##|  |  |  |result: one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16], pps = [in]
##|  |  |result: one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16], pps = [in]
##|  |result: one d15: Milk | one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16] && d15.drink_house=h16, pps = []
one d15: Milk | one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16] && d15.drink_house=h16

The Norwegian lives next to the blue house .
(S (NP The Norwegian) (VP lives (PP next to (NP the blue house))) .)
##|  Pattern: _visit_S__NP_VP, Invoking: _visit_ANY__NP_VP
##|  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |result: one c30: Norwegian | 0==0, pps = []
##|  |  Pattern: _visit_VP__PP, Invoking: _visit_VP__ANNY
##|  |  |  Pattern: _visit_PP__NP, Invoking: _visit_PP__ANY
##|  |  |  |  Pattern: _visit_NP__, Invoking: _visit_NP__
##|  |  |  |  |result: one h31: House | h31.~color_house=Blue, pps = []
##|  |  |  |result: one h31: House | h31.~color_house=Blue, pps = [next, to]
##|  |  |result: one h31: House | h31.~color_house=Blue, pps = [next, to]
##|  |result: one c30: Norwegian | one h31: House | h31.~color_house=Blue && c30.citizen_house in h31.next_to, pps = []
one c30: Norwegian | one h31: House | h31.~color_house=Blue && c30.citizen_house in h31.next_to
```

**Fig. 4.** Translation trace for some of the sentences from the Zebra puzzle

The original Einstein puzzle contains precisely the same text as the original publication in Life magazine. The modified version contains eliminates ambiguity, while the randomly generated puzzle was taken from the website `http://www.mensus.net/brain/logic.shtml`. The generalized semantic translation rules were not able to correctly translate the original puzzle, making modifications to eliminate ambiguity necessary. To allow PUZZLER to solve the original formulation of the puzzle, we added special rules to handle the ambiguous sentences. The solution for the original formulation is shown in Figure 5. The randomly generated puzzle was taken directly from the website, and our generalized rules were able to correctly infer its correct solution.

There are still some puzzles that PUZZLER cannot solve. As we observed with the randomly generated puzzle, however, it is relatively straightforward to extend our framework to solve new types of puzzles.

## 7 Future Work

We briefly describe some ways to extend and improve on the work we have presented.

### 7.1 Queries

Our current solution produces instances consistent with the set of constraints represented by a logic puzzle—that is, a solution. To answer questions like who owns the zebra, however, the user currently must examine the resulting instance manually. This compromise is reasonable, since examining the instance is not difficult and yields far more information than the answer to a single question. However, it is also desirable to be able to answer the queries directly.

Parsing these queries is straightforward, and can be accomplished using our existing parsing strategy. The semantic translation presents a more difficult task, as new types of background information are required. The word who, for example, expresses a query over a particular type—that of people. This kind of type inference is difficult, since a particular puzzle may contain many types that could apply; picking the correct one requires background knowledge.

Given an appropriate Alloy encoding, however, extracting the desired answer from the Alloy Analyzer should not be difficult: given the right background information, the query can be translated directly to a constraint that instantiates a special variable to the appropriate result. The solution instance, then, would contain an assignment for that variable representing the answer to the query. Transforming this solution into an English-language answer would be more difficult. The process of translation into logical constraints results in the loss of language-specific information vital to the reconstruction of a natural-language answer. Given the assignment $x = \mathsf{Englishman}$, for example, we would like the answer The Englishman. Even this simple transformation requires the addition of a determiner that was present in the original puzzle but was eliminated during the semantic translation. Moreover, producing such natural-language answers is a difficult problem in itself.

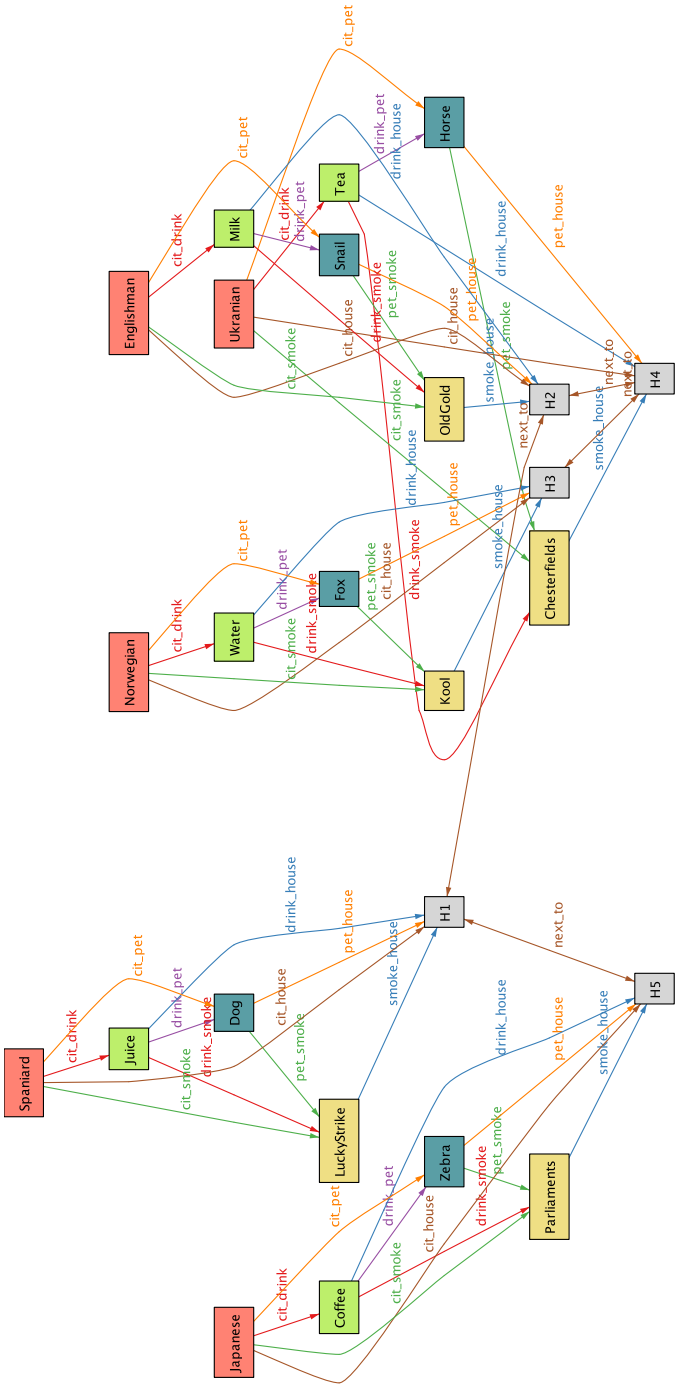


**Fig. 5.** Alloy solution for the Zebra puzzle

### 7.2 Background Information

Our solution requires the user to enumerate the domains of a problem. As we have discussed, inferring these domains is a very difficult task, but our inability to do so means that our analysis is not completely automatic.

Currently, we use WordNet [3] to find canonical forms of nouns (transforming plural nouns to their singular forms, for example). But WordNet also provides the ability to categorize words—a capability that could be used to automatically infer the types present in a problem. WordNet associates English words with hierarchies representing categories to which the word belongs, moving from most specific to most general. Given a set of words, then, it is possible to use WordNet to find the most specific category to which all words in the set belong.

This facility could be used to make an educated guess at the types present in a given problem. Most logic puzzles consist of a set of people, each of whom is mapped to a single member of each other type in the puzzle. While this is not always the case, it can generally be used to determine the size of each type; once these sizes are fixed, different possibilities for the sets of members of each type can be given to WordNet, and the possibility yielding the most specific set of categories will most likely be correct. These categories become the set of types of the problem.

This solution may be difficult to apply in general, because it relies on assumptions that may not always hold. The number of members of each type, for example, may not always be the same; even worse, the permutation of members yielding the most specific categorization may not actually be correct. Both of these situations could cause incorrect answers to be given without notification that something is wrong. A smart set of heuristics could make this possibility rare, but building such a set of heuristics was beyond the scope of this project.

### 7.3 Other Applications

In this work, we have treated logic puzzles as an appropriate starting point for automatic analysis of natural language in many domains that require similar background information inference. Many such domains exist, and, interestingly, are characterized well by logic puzzles. That is, the problems we have tackled are precisely those that are relevant to other domains: many require logical reasoning in addition to information extraction, and most require "human intuition" in the form of background knowledge.

Security policies, for example, are often written in natural language; interpreting them has traditionally been the job of humans. Using our technique, such policies could be interpreted automatically, and logically sound analysis could be performed to determine whether or not the security policy has been broken.

Applying our tools to these domains should be relatively straightforward, since we have designed them to be as general as possible. We expect that some domain-specific changes will be necessary to make our techniques effective in some domains, but that these changes will be reasonable.

## 8 Related Work

Scwhitter [6] assumes the informal description of the puzzle to be generated from a controlled natural language rather than a more general description grammar; this assumption helps in making the logical content of the puzzle explicit and machine-processable. Our approach does not restrict the grammar of the puzzle description language and therefore is much more general. Schwitter says that there exists **no** natural language processing system that can take the original version of Einstein's Riddle as input and infer the correct solution automatically. While PUZZLER does not solve the puzzle completely on its own, it represents the next step forward towards automating the solution.

Lev et. all [5] present a similar system which tries to infer solutions from the logic puzzles. The logic puzzles they consider are multiple choice questions of the sort found in the Law School Admission Test (LSAT) and the old analytic section of the Graduate Record Exam (GRE). The Zebra puzzles are usually much larger than them, and also the answers to them are not known apriori unlike the case of multiple choice questions. The multiple choice options can be used to perform consistency check with the question description whereas in Zebra puzzles the solution needs to be inferred only from the description itself.

## 9 Conclusions

We have presented PUZZLER, a mostly-automated solver for logic puzzles in the style of the Einstein puzzle. Unlike previous approaches, our tool requires only the set of domains present in a given puzzle, and infers the remaining background information. This results in possibly unsound inferences, but these inferences are often required to solve logic puzzles; PUZZLER takes a conservative approach to background information to avoid making mistakes. We expect a similar approach to be applicable to many other NLP domains that require background knowledge.

## References


1. Life international: Who owns the zebra? *Life International magazine 17,95*, December, 1962.
2. M. Collins. Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29(4):589–637, 2003.
3. C. Fellbaum. *Wordnet: An Electronic Lexical Database*. MIT Press, 1998.
4. D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.
5. I. Lev, B. MacCartney, C. D. Manning, and R. Levy. Solving logic puzzles: From robust processing to precise semantics. In *Proceedings of the 2nd Workshop on Text Meaning and Interpretation at ACL 2004*, pages 9–16, 2004.
6. R. Schwitter. Reconstructing hard problems in a human-readable and machine-processable way. In *PRICAI*, pages 1046–1052, 2008.

7. D. D. K. Sleator and D. Temperley. Parsing english with a link grammar. In *Third International Workshop on Parsing Technologies*, 1991.
8. E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

## A Complete Model Listing

For completeness, we list the complete Alloy model generated by PUZZLER for the original Einstein puzzle.

```
/* ****************************************************************************
/*  Automatically generated at Sat May 16 11:41:06 EDT 2009
/*  @author AlloyPuzzler
/* ****************************************************************************/

open util/ordering[House] as houseOrd

abstract sig Citizen {
  citizen_drink : one Drink,
  citizen_pet : one Pet,
  citizen_smoke : one Smoke,
  citizen_color : one Color,
  citizen_house : one House,
} {
}

one sig Englishman extends Citizen {}
one sig Spaniard extends Citizen {}
one sig Norwegian extends Citizen {}
one sig Japanese extends Citizen {}
one sig Ukrainian extends Citizen {}


abstract sig Drink {
  drink_pet : one Pet,
  drink_smoke : one Smoke,
  drink_color : one Color,
  drink_house : one House,
} {
  one this.~citizen_drink
  one x: Citizen | x.citizen_drink = this ∧
    x.citizen_pet=this.@drink_pet ∧
    x.citizen_smoke=this.@drink_smoke ∧
    x.citizen_color=this.@drink_color ∧
    x.citizen_house=this.@drink_house
}

one sig Coffee extends Drink {}
one sig Milk extends Drink {}
one sig Orange_Juice extends Drink {}
one sig Water extends Drink {}
one sig Tea extends Drink {}


abstract sig Pet {
  pet_smoke : one Smoke,
  pet_color : one Color,
  pet_house : one House,
} {
  one this.~citizen_pet
  one this.~drink_pet
  one x: Citizen | x.citizen_pet = this ∧
    x.citizen_smoke=this.@pet_smoke ∧
```

```
54      x.citizen_color=this.@pet_color ∧
55      x.citizen_house=this.@pet_house
56  }
57
58  one sig Dog extends Pet {}
59  one sig Snail extends Pet {}
60  one sig Horse extends Pet {}
61  one sig Zebra extends Pet {}
62  one sig Fox extends Pet {}
63
64
65  abstract sig Smoke {
66    smoke_color : one Color,
67    smoke_house : one House,
68  } {
69    one this.~citizen_smoke
70    one this.~drink_smoke
71    one this.~pet_smoke
72    one x: Citizen | x.citizen_smoke = this ∧
73      x.citizen_color=this.@smoke_color ∧
74      x.citizen_house=this.@smoke_house
75  }
76
77  one sig Old_Gold extends Smoke {}
78  one sig Kool extends Smoke {}
79  one sig Chesterfields extends Smoke {}
80  one sig Lucky_Strike extends Smoke {}
81  one sig Parliaments extends Smoke {}
82
83
84  abstract sig Color {
85    color_house : one House,
86  } {
87    one this.~citizen_color
88    one this.~drink_color
89    one this.~pet_color
90    one this.~smoke_color
91    one x: Citizen | x.citizen_color = this ∧
92      x.citizen_house=this.@color_house
93  }
94
95  one sig Red extends Color {}
96  one sig Green extends Color {}
97  one sig Ivory extends Color {}
98  one sig Yellow extends Color {}
99  one sig Blue extends Color {}
100
101
102 abstract sig House {
103   next_to : set House,
104 } {
105   one this.~citizen_house
106   one this.~drink_house
107   one this.~pet_house
108   one this.~smoke_house
109   one this.~color_house
110   one x: Citizen | x.citizen_house = this
111   this.@next_to = { houseOrd/next[this] ∪ houseOrd/prev[this] }
112 }
113
114 one sig H1 extends House {}
115 one sig H2 extends House {}
116 one sig H3 extends House {}
117 one sig H4 extends House {}
118 one sig H5 extends House {}
119
120
121 fact f_House_ordering {
```

```
122    houseOrd/first=H1
123    houseOrd/next[H1]=H2
124    houseOrd/next[H2]=H3
125    houseOrd/next[H3]=H4
126    houseOrd/next[H4]=H5
127 }
128
129 -- There are five houses .
130 fact f0 {
131    one h0: House | #houseOrd/prevs[h0]=4
132 }
133
134 -- The Englishman lives in the red house .
135 fact f1 {
136    one c1: Englishman | one h2: House | h2.~color_house=Red ∧
    c1.citizen_house=h2
137 }
138
139 -- The Spaniard owns the dog .
140 fact f2 {
141    one c3: Spaniard | one p4: Dog | c3.citizen_pet=p4
142 }
143
144 -- Coffee is drunk in the green house .
145 fact f3 {
146    one d5: Coffee | one h6: House | h6.~color_house=Green ∧
    d5.drink_house=h6
147 }
148
149 -- The Ukrainian drinks tea .
150 fact f4 {
151    one c7: Ukrainian | one d8: Tea | c7.citizen_drink=d8
152 }
153
154 -- The green house is immediately to the right of the ivory house .
155 fact f5 {
156    one h9: House | h9.~color_house=Green ∧ one h10: House | h10.~color_house=Ivory ∧
    houseOrd/prev[h9] = h10
157 }
158
159 -- The Old_Gold smoker owns snails .
160 fact f6 {
161    one s11: Old_Gold | one p12: Snail | s11.~pet_smoke=p12
162 }
163
164 -- Kools are smoked in the yellow house .
165 fact f7 {
166    one s13: Kool | one h14: House | h14.~color_house=Yellow ∧
    s13.smoke_house=h14
167 }
168
169 -- Milk is drunk in the middle house .
170 fact f8 {
171    one d15: Milk | one h16: House | #houseOrd/prevs[h16]=#houseOrd/nexts[h16] ∧
    d15.drink_house=h16
172 }
173
174 -- The Norwegian lives in the first house .
175 fact f9 {
176    one c17: Norwegian | one h18: House | houseOrd/first=h18 ∧
    c17.citizen_house=h18
177 }
178
179 -- The man who smokes Chesterfields lives in the house which is next to the man who has the fox .
180 fact f10 {
181    one s19: Chesterfields | one h20: House | one p21: Fox | h20 in p21.pet_house.next_to ∧
    s19.smoke_house=h20
182 }
```

```
183
184 -- Kools are smoked in the house which is next to the house in which the horse is kept .
185 fact f11 {
186   one s22: Kool | one h23: House | one p24: Horse | h23 in p24.pet_house.next_to ∧
    s22.smoke_house=h23
187 }
188
189 -- The Lucky_Strike smoker drinks Orange_Juice .
190 fact f12 {
191   one s25: Lucky_Strike | one d26: Orange_Juice | s25.~drink_smoke=d26
192 }
193
194 -- The Japanese smokes Parliaments .
195 fact f13 {
196   one c27: Japanese | one s28: Parliaments | c27.citizen_smoke=s28
197 }
198
199 -- The Norwegian lives next to the blue house .
200 fact f14 {
201   one c29: Norwegian | one h30: House | h30.~color_house=Blue ∧
    c29.citizen_house in h30.next_to
202 }
203
204 pred solve {}
205
206 run solve for 5
```